Hoare triples

Floyd-Hoare Logic, Separation Logic
1. Floyd-Hoare Logic 1969

Reasoning about control
Hoare triples

- \{A\} \ p \ \{B\} a Hoare triple

partial correctness:

if the initial state satisfies assertion \(A\), and if the execution of program \(p\) terminates, then the final state satisfies assertion \(B\)

- inference rules

\[
\begin{align*}
\{A[a/X]\}X := a \{A\} & \quad \{A\} \text{skip} \{A\} & \quad \{A_1\} \ p_1 \ \{A_2\} \quad \{A_2\} \ p_2 \ \{A_3\} \\
\{A \wedge a \geq 0\} \ p_1 \ \{B\} & \quad \{A \wedge \neg(a \geq 0)\} \ p_2 \ \{B\} \\
\{A\} \text{if } a \geq 0 \text{ then } p_1 \text{ else } p_2 \ \{B\} \\
\{A_I \wedge a \geq 0\} \ p \ \{A_I\} \\
\{A_I\} \text{while } a \geq 0 \text{ do } p \ \{A_I \wedge \neg(a \geq 0)\} \\
A_1 \Rightarrow A_2 & \quad \{A_2\} \ p \ \{B_2\} & \quad B_2 \Rightarrow B_1 \\
\{A_1\} \ p \ \{B_1\}
\end{align*}
\]

- expressive properties

*functional correctness* rather than absence of runtime errors
Hoare logic — main ingredients

**programmers**

\[ X := Y + 3 \]

**(Hoare) logicians**

\[ X \geq Y + 3 \]

**ingredients in Hoare logic:**

1. a language for programs \( p \)
2. a language for assertions \( A \)
3. inference rules

**important aspects:**

- invariants in loops
- logical deduction rule
- backward reasoning (in the rule for assignment)
Hoare logic: metatheoretical properties

- operational semantics and validity
  - big step operational semantics for $\text{IMP}$: $\sigma, p \downarrow \sigma'$
    - $\sigma$ is an environment
    - $\sigma : V \to \mathbb{Z}$ a map from variables to integers
      given some program $p$, $\sigma$ is a partial mapping from a finite set of variables to $\mathbb{Z}$
  - the triple $\{A\} p \{B\}$ is valid:
    for all $\sigma$, if $\sigma$ satisfies $A$ and $\sigma, p \downarrow \sigma'$, then $\sigma'$ satisfies $B$
Hoare logic: metatheoretical properties

- operational semantics and validity
  - big step operational semantics for IMP: $\sigma, p \downarrow \sigma'$
    - $\sigma$ is an environment
    - $\sigma : \mathcal{V} \rightarrow \mathbb{Z}$ a map from variables to integers
      given some program $p$, $\sigma$ is a partial mapping from a *finite set of variables* to $\mathbb{Z}$
    - the triple $\{A\} p \{B\}$ is valid:
      for all $\sigma$, if $\sigma$ satisfies $A$ and $\sigma, p \downarrow \sigma'$, then $\sigma'$ satisfies $B$
  - correctness
    - If the triple $\{A\} p \{B\}$ can be derived using the inference rules of Hoare logic, then it is valid.
      - NB: we could also rely on *denotational semantics*
        associate to each program $p$ some function $F_p$ from environments to environments
  - (relative) completeness
    - any valid triple can be constructed in Hoare logic, *provided* we can decide validity of the assertions (*i.e.*, decide whether $A$ always holds)
    - logic rules capture the properties we want to express
Correct rules and completeness

- the 6 rules of Hoare logic are not the only correct rules
- for instance, the rule of constancy is correct too

$$\begin{align*}
\{A\} \ p \ \{B\} \\
\{A \land C\} \ p \ \{B \land C\}
\end{align*}$$

no variable in $C$ is modified by $p$

- completeness: no new Hoare triple can be established if we add the rule of constancy
  - the 6 rules “tell everything”
  - using the rule of constancy makes proofs easier/more natural/more readable

somehow, completeness is not only a theoretical question
The axiom for assignment

the axiom for assignment goes backwards

\[
\{A[a/X]\} \ X := a \ {A}
\]

(consider \(X := X + 3\) to convince yourself)

Floyd’s **forward axiom**

\[
\{A\} \ X := a \ {\exists i. (X = a[i/X] \land A[i/X])}\}
\]
The axiom for assignment

does backwards

\[
\{A[a/X]\} \ X := a \{A\}
\]

(consider \(X := X + 3\) to convince yourself)

Floyd’s forward axiom

\[
\{A\} \ X := a \{\exists i. (X = a[i/X] \land A[i/X])\}
\]

see also

\[
\{A \land X = i\} \ X := a \{A[i/X] \land X = a[i/X]\}
\]

\(i:\) “ghost variable” (should probably be written \(I\)
Synonyms

assertion formula

environment store

correctness soundness

A

σ

for a rule
2. Separation Logic \( \sim 2000 \)

Reasoning about memory
Programs manipulating pointers

- Hoare logic deals essentially with **control**
  
  \[
  \text{if } a \geq 0 \text{ then } p_1 \text{ else } p_2 \quad p_1; p_2 \quad \text{while } a \geq 0 \text{ do } p
  \]

- move to a **richer language**: add (some kind of) pointers and handling of **memory**
  - allocation
  - modification (move pointers around)
  - liberation/deallocation

- different kinds of properties
  - typical runtime errors we want to detect:
    - memory leaks, invalid disposal, invalid accesses
      - typically, other approaches either **assume** memory safety, or forbid dynamic memory allocation
  - describe what programs manipulating pointers do

- adopt the same methodological framework

  Separation Logic is an enrichment of Floyd-Hoare logic
Extending IMP

structure of memory at runtime

in (traditional) Hoare-Floyd logic, programs manipulate variables

the environment just records the (integer) value of each variable

that is all we know about the memory

dynamically allocated memory: add a heap component

extending the programming language

on the board
Extending IMP

- structure of memory at runtime
  - in (traditional) Hoare-Floyd logic, programs manipulate variables
    - the environment just records the (integer) value of each variable
      that is all we know about the memory
  - dynamically allocated memory: add a heap component
  - extending the programming language

what does this program do?

```
J := nil;
while I != nil do
  K := [I + 1];
  [I + 1] := J;
  J := I;
  I := K
```
Extending assertions: introducing *heap formulas*

- a memory state is \((\sigma, h)\) where
  - \(\sigma\) is a store
  - \(h\) is a heap

- Hoare logic assertions state properties about the environment
  \[ X \geq Y \cdot Z + Q \quad \land \quad T > 0 \]

- add formulas to reason about the heap

- NB: \(X \mapsto 52\) usually makes more sense than \(32 \mapsto 52\) (both are assertions)
Hoare triples in Separation logic — interpretation

\{A\} p \{B\} holds iff

\forall \sigma, h., if (\sigma, h) \models A, ((\sigma, h) \text{ satisfies } A)
then

- (\sigma, h), p \not\Downarrow \text{error}, and
- if (\sigma, h), p \Downarrow (\sigma', h'), then (\sigma', h') \models B

like in traditional Hoare logic, but:

- the state has a heap component
- absence of forbidden access to the memory
Small axioms

axioms for heap-accessing operations are tight
they only refer to the part of the heap they need to access (their footprint)

along these lines, tight version of the axiom for (usual) assignment:

\[
\{ X = i \land \text{emp} \} X := a \{ X = a \land \text{emp} \}
\]

if \( X \) does not occur in \( a \), the rule becomes simpler:

\[
\{ \text{emp} \} X := a \{ X = a \land \text{emp} \}
\]

moreover, being tight tells us the following:

suppose we can prove \( \{ 10 \mapsto 32 \} p \{ 10 \mapsto 52 \} \)

whatever \( p \) is

then we know that if we run \( p \) in a state where cell 11 is allocated,
then \( p \) will not change the value of 11
Axioms for heap-accessing operations are **tight**
they only refer to the part of the heap they need to access (their **footprint**).

Along these lines,
tight version of the axiom for (usual) assignment:

\[
\{ X = i \land \text{emp} \} \ X := a \ \{ X = a[i/X] \land \text{emp} \} \\
\]

If \( X \) does not occur in \( a \),
the rule becomes simpler:

\[
\{ \text{emp} \} \ X := a \ \{ X = a \land \text{emp} \}
\]
Small axioms

- Axioms for heap-accessing operations are **tight**
  - They only refer to the part of the heap they need to access (their footprint).

- Along these lines, the tight version of the axiom for (usual) assignment:

  \[
  \{X = i \land emp\} X := a \{X = a[i/X] \land emp\}
  \]

  If \(X\) does not occur in \(a\), the rule becomes simpler:

  \[
  \{emp\} X := a \{X = a \land emp\}
  \]

- Moreover, being tight tells us the following:
  - Suppose we can prove \(\{10 \mapsto 32\} p \{10 \mapsto 52\}\) whatever \(p\) is.
  - Then we know that if we run \(p\) in a state where cell 11 is allocated, then \(p\) will not change the value of 11.
The frame rule

- the rules of Hoare logic remain sound
- the rule of consistency becomes unsound

\[
\begin{align*}
\frac{\{A\} p \{B\}}{\{A \land C\} p \{B \land C\}} & \quad \text{no variable in } C \\
\text{is modified by } p
\end{align*}
\]

\[
\begin{align*}
\{x \mapsto \_\} [x] := 4 \{x \mapsto 4\} \\
\{x \mapsto \_ \land y \mapsto 3\} [x] := 4 \{x \mapsto 4 \land y \mapsto 3\}
\end{align*}
\]

what if \(x = y\)?

- separation logic is inherently modular as opposed to whole program verification
The frame rule

- the rules of Hoare logic remain sound
- the rule of consistency becomes unsound
  \[ \frac{\{A\} p \{B\}}{\{A \land C\} p \{B \land C\}} \]
  no variable in \( C \) is modified by \( p \)

\[
\begin{align*}
\{x \mapsto \_\} [x] := 4 \{x \mapsto 4\} \\
\{x \mapsto \_ \land y \mapsto 3\} [x] := 4 \{x \mapsto 4 \land y \mapsto 3\}
\end{align*}
\]

what if \( x = y \)?

- the Frame Rule
  \[ \frac{\{A\} p \{B\}}{\{A \ast C\} p \{B \ast C\}} \]
  no variable in \( C \) is modified by \( p \)

- separation logic is inherently modular as opposed to whole program verification
Separation logic: sum up

- inference rules
  - those of Hoare logic
  - those for the new programming constructs

- important things:
  - invariants in while loops, backward rule for assignment, consequence rule
  - (tight) small axioms, footprint, frame rule

- metatheoretical properties
  - correctness
  - completeness
Beyond absence of runtime errors:

recursive data structures
Reasoning about lists

- a linked list in memory is something like

\[(X_1 \mapsto k_1, X_2) \ast (X_2 \mapsto k_2, X_3) \ast \cdots \ast (X_n \mapsto k_n, \text{nil})\]

\[(X \mapsto a, b) \text{ stands for } X \mapsto a \ast (X + 1) \mapsto b\]

- describe the structure using assertions:

  add the possibility to write (recursive) equations

\[
\text{list}(i) \quad = \quad (i = \text{nil} \land \text{emp}) \lor (\exists j, k. (i \mapsto k, j) \ast \text{list}(j))
\]
Reasoning about lists

- a linked list in memory is something like

\[
(X_1 \mapsto k_1, X_2) \ast (X_2 \mapsto k_2, X_3) \ast \cdots \ast (X_n \mapsto k_n, \text{nil})
\]

\[(X \mapsto a, b) \text{ stands for } X \mapsto a \ast (X + 1) \mapsto b\]

- describe the structure using assertions:

add the possibility to write \textbf{(recursive) equations}

\[
\text{list}(i) = (i = \text{nil} \land \text{emp}) \lor (\exists j, k. (i \mapsto k, j) \ast \text{list}(j))
\]

- the formula above just specifies that we have a list in memory

we can rely on “mathematical lists” \([\text{[]} , k::ks\]) to provide a
more informative definition

\[
\text{list}(\text{[]}, i) = \text{emp} \land i = \text{nil}
\]

\[
\text{list}(k::ks, i) = \exists j. (i \mapsto k, j) \ast \text{list}(ks, j)
\]
Recursive data structures

- we can specify similarly various kinds of data structures
- we can give a meaning to such recursive definitions using Tarski’s theorem
Recursive data structures

- we can specify similarly various kinds of data structures
- we can give a meaning to such recursive definitions using Tarski’s theorem

- an exercise

\[
\text{list}(i) = (i = \text{nil} \land \text{emp}) \lor (\exists j, k. (i \mapsto k, j) \ast \text{list}(j))
\]

- write the code for a while loop that deallocates a linked list,
- and prove \( \{\text{list}(X)\} p \{\text{emp}\} \), where \( p \) is your program
On the weirdness of auxiliary variables

- in the lecture we saw the small axiom for lookup
  \[ \{ a \mapsto i \land X = j \} X := [a] \{ X = i \land a[j/X] \mapsto i \} \]

- in the TD you saw its simpler form
  \[ \{ a \mapsto i \} X := [a] \{ X = i \land a[j/X] \mapsto i \} \] if \( X \) does not appear in \( a \)

- how does one entail the other?
  rule of auxiliary variable elimination
  \[ \frac{\{A\} \ p \ \{B\}}{\exists u. (\{A\} \ p \ \{B\})} \]
  if \( u \) does not appear in \( p \)

- if \( X \) does not appear in \( a \), \( a[j/X] = a \)

  moreover,
  \[ \frac{\exists j. (a \mapsto i \land X = j)) \} \ X := [a] \{ \exists j. (X = i \land a \mapsto i) \} \]
  \[ \iff a \mapsto i \land \exists j. X = j \]
  \[ \iff a \mapsto i \]
  \[ \iff X = i \land a \mapsto i \]
Going further
Reasoning about concurrent programs

concurrent separation logic

- shared memory, several threads
- permissions, locks, critical sections
- ownership
Towards automation

- Hoare logic and separation logic are used naturally in an interactive manner
- if loop invariants are provided (as well as the global pre and post conditions), we can automatically chop the verification task into the proof of slices of the form

\[
\{A\} \ p_1; p_2; \ldots; p_k \ {B},
\]

where the \( p_i \)s are elementary commands.
- construction of the Hoare triple boils down to being able to prove *entailments* between assertions, \( A \vdash B \)

cf. the *Why3 tool* (Filliâtre et al.)
Towards automation of separation logic

- restrict the set of possible formulas: **symbolic heaps**

  \[ P = P_1 \land \cdots \land P_k \quad \text{pure formulas} \]

  \[ P \land H \quad H = H_1 \land \cdots \land H_n \quad \text{simple heap formulas} \]

  (for instance, \( \mapsto \), \text{list}, emp)

- small axioms are adapted to symbolic heaps, yielding a “specialised operational semantics”

- deciding entailments
  - for the **pure part** of symbolic heaps, standard approaches (theorem provers/automatic decision procedures)
  - for the **heap components**, exploiting implications like
    - \((\text{list}(i) \land i = \text{nil}) \Rightarrow \text{emp}\)
    - \((i \mapsto k, j \land \text{list}(j)) \Rightarrow \text{list}(i)\)

- more automation: **discovering loop invariants**
  - back to **abstract interpretation**: abstract execution, generating a postcondition as we run through the loop
  - sometimes **abstracting** (narrowing) to insure termination
    - e.g., replacing \(i \mapsto k, j \land j \mapsto (k', j') \land j' = \text{nil}\) with \text{list}(i)
      (loosing information about the size of the list)
Modular analysis

- use the automated framework to analyse **functions manipulating pointers**
- compute Hoare triples for functions, *without information about the rest of the code*
- solve \( A \ast ?\text{antiframe} \vdash B \ast ?\text{frame} \)
  - **antiframe**: missing portion of heap because of function calls, the outer function body should have some parts of the heap in its **precondition**
  - **frame**: leftover portion of heap the **postcondition** of the outer function body specifies what parts of the heap are left unchanged